RL-TR-94-67
Final Technical Report
June 1994

COMMUNICATION NETWORK SOFTWARE ANALYSIS

Clarkson University and Boston University

Robert A. Meyer, David A. Perreault

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

Rome Laboratory
Air Force Materiel Command
Griffiss Air Force Base, New York

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

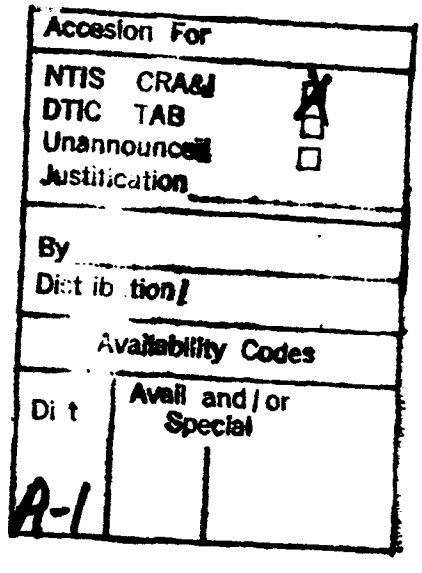
RL-TR-94-67 has been reviewed and is approved for publication.

APPROVED: Charles Meyer

CHARLES MEYER
Project Engineer

FOR THE COMMANDER John A Graniero

JOHN A. GRANIERO
Chief Scientist for C3



If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL (C3BC) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	June 1994	Final Jun 92 - Dec 93

4. TITLE AND SUBTITLE
COMMUNICATION NETWORK SOFTWARE ANALYSIS

5. FUNDING NUMBERS
C - F30602-92-C-0083
PE - 33126F
PR - 2022
TA - 05
WU - P1

6. AUTHOR(S)
Robert A. Meyer
David A. Perreault

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Clarkson University (Electrical & Computer Engr Dept)
Potsdam NY 13699
Boston University (Microprocessor Research Laboratory)
Boston MA 02215

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Rome Laboratory (C3BC)
525 Brooks Road
Griffiss AFB NY 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
RL-TR-94-67

11. SUPPLEMENTARY NOTES
Rome Laboratory Project Engineer: Charles Meyer/C3BC/(315) 330-1880

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)
Synchronous (or simultaneous) program execution (SPE) is a technique that allows the same program to run synchronously on geographically separated computers. However, the results generated appear to the users at the different sites as if the users are sitting in front of the same computer. The difference between the SPE technique and the distributed systems techniques is that with the SPE technique the same execution takes place at each computer, while with distributed systems techniques, parts of one program execute on different computers. The SPE technique can be used in reducing communications in cases such as computer conferencing.
The SPE technique is based on the creation of a Shell running at each computer and the transmission of messages between the Shells. Only a low bandwidth line connects the computer systems. The Shell is constructed at each computer system, in software, and resides between the operating system and the executing program and between the entering inputs and the operating system.
This paper presents and analyzes Petri nets that model the synchronization of simultaneous program execution between two computers. The analysis of the Petri nets indicates proper operation of the system.

14. SUBJECT TERMS
Shell, Petri Nets, Computers

15. NUMBER OF PAGES
20

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Communication Network Software Analysis

EXECUTIVE SUMMARY

Synchronous (or simultaneous) program execution (SPE) is a technique that allows the same program to run synchronously on geographically separated computers. However, the results generated appear to the users at the different sites as if the users are sitting in front of the same computer. The difference between the SPE technique and the distributed systems techniques is that with the SPE technique the same execution takes place at each computer , while with distributed systems techniques, parts of one program execute on different computers. The SPE technique can be used in reducing communications in cases such as computer conferencing.

Synchronized execution means that the programs must have the "same execution" and be closely synchronized. In order for the programs to have same execution, they must get the same input at the same point in their execution. Inputs are also used to synchronize the executions of the programs. The inputs can originate on any computer and get distributed to the other computers. Input accessing methods are required to specify which computer's inputs are valid at any given time.

The SPE technique is based on the creation of a Shell running at each computer and the transmission of messages between the Shells. Only a low bandwidth line connects the computer systems. The Shell is constructed at each computer system, in software, and resides between the operating system and the executing program and between the entering inputs and the operating system.

This paper presents and analyzes Petri nets that model the synchronization of simultaneous program execution between two computers. The interactions between the Shell, the operating system and the application program of the two computers, as well as the messages sent between them are modeled with Petri nets. The analysis of the Petri nets indicates proper operation of the system. The analysis includes safeness, conservability and liveness. Programs may execute on computers of different speeds. This results in a delay in the execution of the faster computer. Timing analysis was performed to calculate the delays and the affect they have on the performance of the system.

1. THE SHELL AND THE INPUTS

The Shell can be considered as an extension to the operating system handling the remote execution section. It accepts all the inputs to the system, checks if valid, and transmits them to one of the computers called the Master. The Master collects all the inputs, places them in order, appends some synchronization information and distributes them to the Slave computer. The Shell is created between the application program and the operating system and between the inputs and the operating system. The Shell:

1. runs on both computers

2. gets activated only by an input interrupt and takes the action shown in 3 or 4 bellow.

3. accepts the inputs and hides them from the operating system. If the inputs are valid, it stores them in Temporary Input Buffers (TIB).

4. accepts all requests for inputs and distributes and synchronizes the inputs that are already in the TIBs if any, one at a time, by communicating with the Shell running at the other computer.

The assumption has been made that programs requesting inputs, and inputs entering programs go through the operating system. This is not a severe restriction since most high level programs in single user systems and all programs in multi user systems are written in this manner.

An application program can request input in a *synchronous* or *asynchronous* manner. The application program can either wait until an input gets entered, in which case the input always enters the program at the same point in its execution and is synchronous, or, proceed on its execution and periodically check for the presence of input, in which case it is asynchronous. The number of times C that the application program checks for the presence of an input (through requests to the operating system), until the input gets entered, designate the exact point in program execution that the input got entered, and is used for the synchronization of the asynchronous inputs between the two computer systems. The Shell updates the values of the counter C each time the application program checks for the presence of

an input. The Shells running on the two computers make sure that the input is presented to the application programs on the same C count.

Asynchronous inputs may be entered from either one of the two computers. An input accessing method is required that designates what inputs are valid from each one of the two computers at any time. For example, the keyboards of the two computers may not be active at the same time. The user can specify the input accessing method that they prefer at the beginning of the session. All the inputs from both computers are presented to the Master where they get validated and ordered. The Master considers all valid inputs as if they were its own inputs and distributes them to the Slave, with some synchronization information. The synchronization information is the count C. When an input gets presented to the application programs on the two computers, the count of the Master, Cm, must be equal to the count of the Slave, Cs.

Synchronous inputs may be entered from either of the two computers in which case they must be distributed to the other computer. Synchronization is not required but validation is. There are cases where synchronous inputs mav originate from data that exists on both computers, like reading of files that exist on both computers. In such cases the application program can itself get the input (through the operating system) without any action to be taken by the Shells. The rest of the paper is about the asynchronous inputs.

2. ASYNCHRONOUS INPUT SYNCHRONIZATION

Petri nets are used for modeling the synchronization of the asynchronous inputs. One of the most important use of Petri nets is the modeling of asynchronous systems, especially ones that experience concurrency, asynchronism and nondeterminism [1]. The system under construction possesses all three conditions. The system experiences concurrency since there is execution taking place at two computers. It experiences asynchronism since some inputs are entered in an asynchronous way. Finally, it experiences nondeterminism, in its execution, since the inputs are entered interactively, which means that the execution takes a path related to the user response. In Petri net language, this is called decision or data-dependency. For the analysis that will follow we adopt the theory, notation and formulation of Coolahan and Roussopoulos[2].

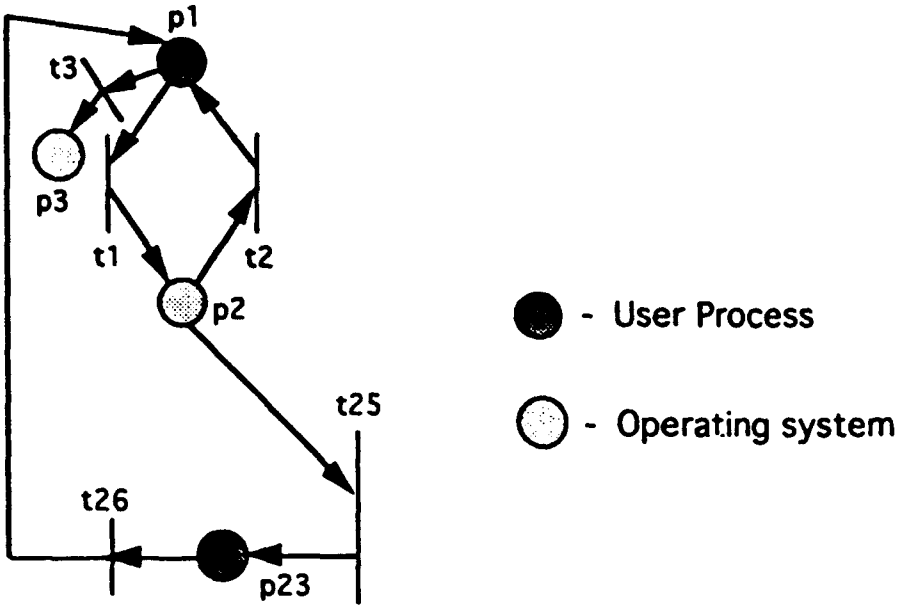


Figure 1 Petri net of User Process checking for inputs

In figure 1 is the Petri net of a running process that checks for an input. Place p1 represents the application process that is executing. When the process wants to check for an input, transition t1 fires and the token goes to place p2. Transition t1 is the call to the operating system to check for an input. In p2 the checking for an input is performed by the operating system. If no input is available, then transition t2 fires and the token returns to p1 where the user process resumes execution. The token loops in p1, t1, p2, t2 until an input is entered. After an input is entered, the next time that the token gets in p2, transition t25 will fire and the loop will terminate. The token goes to p23 where the user process gets the input. Then transition t26 fires and the token returns to p1 where the user process returns to processing. This scenario will continue as long as the process is executing. When the process terminates, transition t3 fires and the token returns to the operating system, place p3.

The handshake between the two computers for synchronizing one asynchronous input is shown in figure 2. If the input is from the Slave, the Slave sends the input to the Master with msg6 and the Slave proceeds on its execution. The Master receives msg6 and checks if the Slave has access of the inputs (e.g. the keyboard, if the input is a key). If it has, the Master accepts the input and considers it as one of its own.

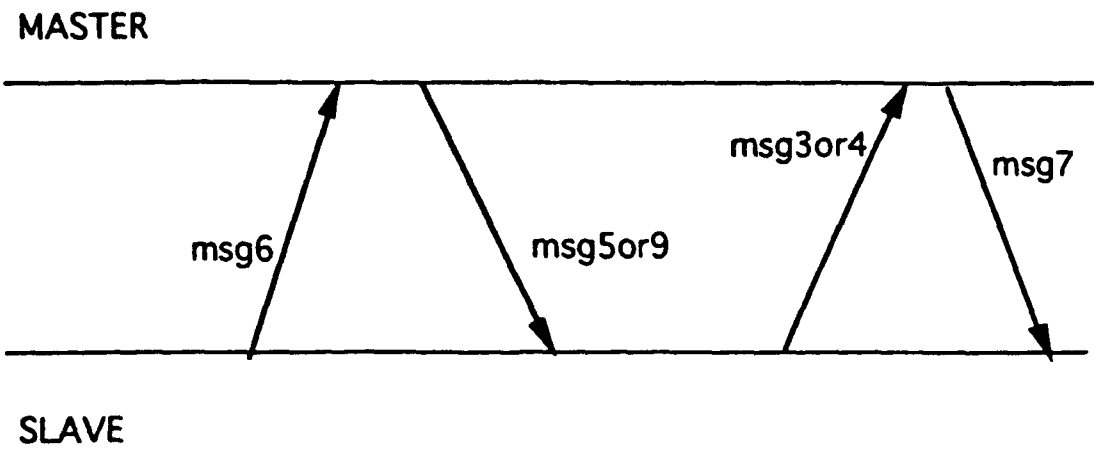


Figure 2 The Handshake for Asynchronous Input Synchronization

If the Master has an input, it sends a msg5 to the Slave with the input and counter, Cm. If that input was a Slave input that arrived at the Master with a msg6, then the message number is 9 instead of 5. When the Slave receives msg5 or msg9 with the input and Cm, it compares Cm with its own count Cs. If Cs<=Cm then the Slave continuous execution to catch up with the Master (Cs=Cm) and then it sends msg3 to the Master. If Cs>Cm then the Slave sends msg4 to the Master with its count Cs and waits. When Master receives msg3 it puts the input in effect, sends msg7 to the Slave and resumes execution. If Master had received msg4, it continues execution until Cm=Cs. Then it puts the input in effect, sends msg7 to the Slave and resumes execution. When Slave receives msg7 it puts the input into effect and resumes execution. The above actions of transmitting and receiving messages, comparing counts, etc, takes place in the Shells running at the Master and the Slave. This is shown in detail in figure 3.

Figure 3 displays the Petri net, which was build based on ideas provided by [3] and [4]. This Perti net is divided into three parts with the dotted lines: the Master, the Slave and the Communication (COMM). This Perti net models the handshake of figure 2. Figure 1 shows how a program requests and gets an input, and figure 3 shows how two programs request and get the same input at the same point in their executions with the help of the Shell. The input accessing method modeled on this Petri net is that the inputs from the Master and the Slave are allowed in the order that they arrive at the Master. No inputs are deleted unless the TIBs local to each machine, get full.The inputs are processed one at a time.

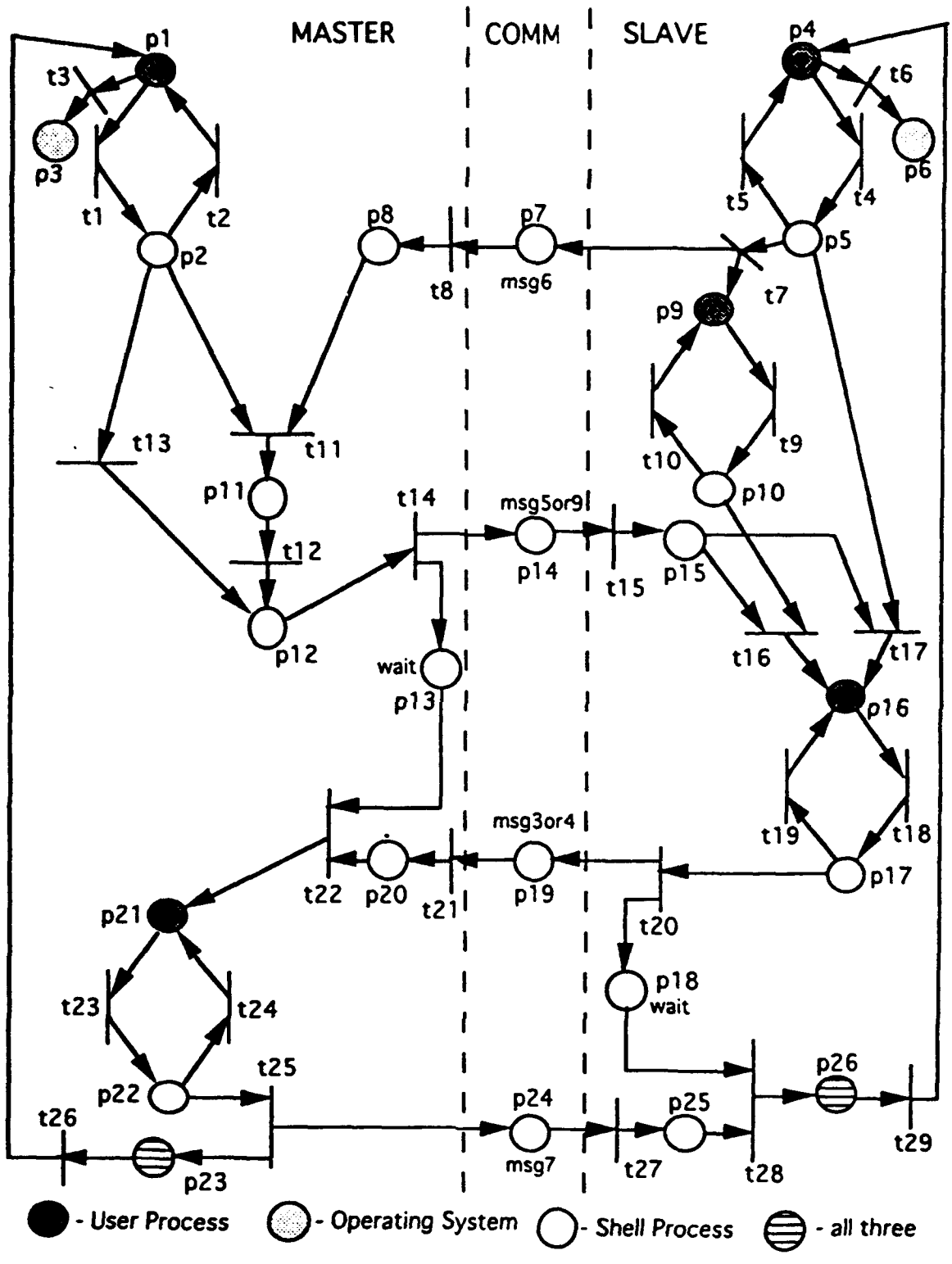


Figure 3 Perti net for synchronizing the asynchronous inputs

The places on the Petri net represent processing or checking of simple conditions. The processing can be part of the application (user) process, of the Shell process, or of the operating system. The transitions on the Petri net represent events happening.

The Petri net of figure 6 consists mainly of: a) five loops b) four messages c) two wait states d) some extra processing states.

The five loops are first: p1, t1, p2, t2; second: p4, t4, p5, t5; third: p9, t9, p10, t10; forth: p16, t18, p17, t19; fifth: p21, t23, p22, t24;

The four messages are: first is "msg6": p7, t8, p8; second is "msg5" or "msg9": p14, t15, p15; third is "msg3" or "msg4" :p19, t21, p20; and forth is "msg7": p24, t27, p25.

Messages consist of three fields: 1. type of message (or msg number) 2. input info 3. local counter. In the case of keyboard inputs, the message type field is one byte long, the input info field is two bytes: one byte for the ASCII code of the key and one byte for the extended code or scan code, and the counter field can be of fixed or variable size. A variable counter field requires the message to have a forth field containing the message length.

The two wait states are: first: t14, p13, t22; second: t20, p18, t28.

In the first loop, in p1, the user process is running and when it checks for input transition t1 fires and the token comes to p2. As was seen in figure 1 this would be a call to the operating system, checking for inputs, but in this case p2 is the Shell which intercepts the calls of the user process to the operating system concerning input. In p2 the shell will check if a msg6 has arrived from the Slave holding an input (token in p8) and then transition t11 would fire. If no msg6 has arrived from Slave then if a local input has been inserted at Master, t13 will fire; else t2 will fire and the token returns to p1 where the Master user process resumes execution. While a token is ready at p2, priority is given first to t11 to fire, then to t13 and last to t2.

All loops work in a similar manner. Looking at these five loops in figure 3, the top place of each loop is execution of the user process, and counter C increments by

one each time the token passes through. The bottom place is execution in the shell that checks conditions and priorities. All actions required to synchronize the inputs are taken while executing in the Shell.

In the second loop, in place p5 priority is given first to t17 to fire if a message has arrived from the Master. If no message from the Master is available, if an Slave input is available the t7 will fire. Otherwise, t5 will fire.

In the third loop, in place p10, priority is given first to t16 to fire if a message from Master has arrived, else t10 will fire.

In the fourth loop, in place p17, t20 will fire if Cs>=Cm+1; else (if Cs<Cm+1) t19 will fire.

In the fifth loop, in place p22, t24 will fire if Cm<Cs; if Cm=Cs t25 will fire.

Example of Execution of the Perti net.

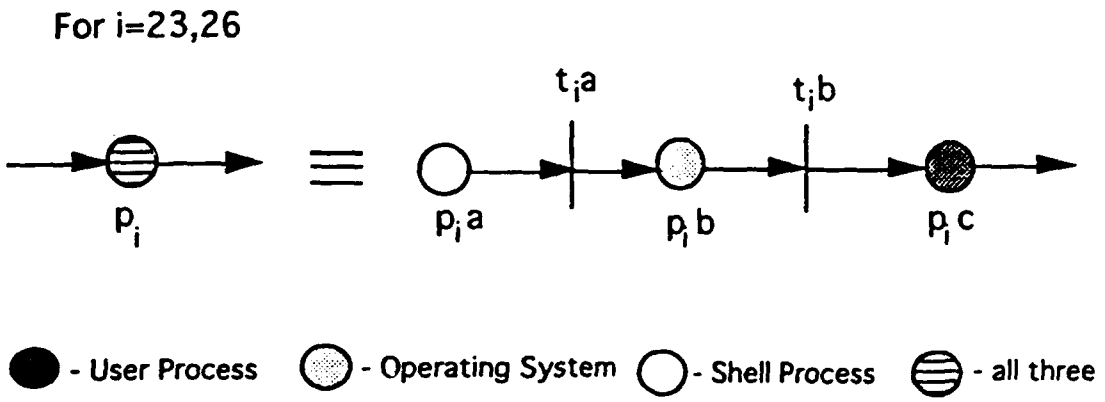
Case 1: Master gets input (key located in one of its TIBs).

The user program starts at p1 for Master and at p4 for the Slave. While Master is waiting for key (Master keyboard TIB is empty), Master is looping at p1, t1, p2, t2 and Slave is looping at p4, t4, p5, t5, Master presses a key (key in TIB placed there by the Shell hiding the key from the operating system). Next time the Master gets at p2, t13 will fire and p12 will prepare msg5 to send to Slave containing the key and Cm. When t14 fires msg5 is send to Slave and the Master waits at p13. In p14 takes place the transmission of msg5 and t15 fires when msg5 arrives in the Slave. In p15 the computer where the Slave process is running receives msg5. The Slave process is executing in the loop p4, t4, p5, t5 and next time the token gets to p5, t17 will be enabled and will fire to p16. At p16 the Slave user process resumes, and the Slave will loop in p16, t18, p17, t19 until Cs>=Cm+1. During this loop the Slave user process has the chance to catch up with the Master user process if necessary. While at p17, t20 will fire and the Slave will wait at p18 while a msg3 (if Cs=Cm+1) or a msg4 (if Cs>Cm+1) is sent to Master. The transmission takes place at p19. When t21 fires, msg3 or msg4 has arrived at Master and is received at p20. A ready token at p20 resumes the wait of the Master at p13. t22 fires and Master user process resumes execution. Master loops at p21, t23, p22, t24 until Cm=Cs. This gives a chance for Master user process to catch up with the Slave user process. While at p22, t25 will

fire (Cm=Cs). A token goes at p23 where the input is passed to the Master user process, Cm is set to zero, and eventually t26 fires and the token returns to the Master user process at p1. Another token is sent to p24 which is msg7 transmitted to the Slave. When t27 fires, msg7 has arrived at Slave and is received at p25. A ready token at p25 resumes the wait of the Salve at p18. When t28 fires the token goes at p26 where the input is passed to the Slave user process, Cs is set to zero, and eventually t29 fires and the token returns to the Slave user process at p4.

Note that the input passed to the Slave user process (p26, t29) could take place during the wait (t20, p18, t28). In other words p26 is executed before p18. Thus the wait is replaced by t20, p26, t29, p18, t28 and the firing of t28 returns the token to p4.

At p23 and p26 the input was passed to the Master or Slave user process respectively. The Shell presents the input, which in this case is a keystroke entered in the keyboard buffer of the Master or the Slave. A call to the operating system is invoked from the Shell to check for an input (key). The call to the operating system returns that an input is present. Since an input is present, a call from the user process to the operating system may follow, to read and process the input. The token returns to p1 or p4 to start processing the next input.



Case2: Slave gets an input (keypress).

Master is looping at p1, t1, p2, t2 and Slave is looping at p4, t4, p5, t5 waiting for a key. When the Slave gets an input, next time the token gets to p5, t7 will fire. A token goes to p7 where msg6 is transmited to Master, and another token goes to p9 making the Slave loop in p9, t9, p10, t10. When msg6 arrives at Master, t8 fires and a

token goes to p8 where the Master receives msg6. The token at p8 will break the loop p1, t1, p2, t2 next time the token comes around to p2. t11 will fire, the token goes to p11 where the Slave's input is considered as Master's input. t12 fires and the token goes to p12 where Master assembles msg9 (instead of msg5 since it is Slave's input) and when t14 fires it transmits it to the Slave. Master is waiting at p13. When msg9 arrives at the Slave, t15 fires and the a token comes to p15 where the Slave receives msg9. Since the Slave is looping at p9, t9, p10, t10, next time it comes around to p10, t16 fires and a token comes to p16. From then on is the same as in Case1 above.

3. PETRI NET ANALYSIS

Analysis on the Perti net was performed to ensure the proper operation of the system. The reachability tree was constructed and studied. Conclusions of the reachability tree are:

1. All the places in the Petri net are safe except place p8 where it is 2-bounded. But that is not a problem since p8 is a buffer that can hold two messages.

Safeness is a property that must hold in order for the system to work properly. Each place represents the execution of a routine. Safeness states: never more than one token at any place. Assume that a token arrives at a place, which means that a routine will start executing. If another token arrives at the same place before the first routine finishes executing, another routine (with the same code) will start executing and the first routine will stop, causing the system to be in an unknown and unwanted state and therefore unable to recover.

2. The Petri net is not conservative since the number of tokens in the net does not remain constant. Even though the number of tokens is not constant, it can be noted that there is one "resident" token at the Master and one at the Slave at all times that show where the Master and the Slave currently execute. In addition, there are some extra tokens generated and deleted upon the transmission and reception of messages that designate processing at communications processors.

3. The Perti net is live, which means that all the transitions are live and that there are not any deadlocks.

The protocol is written is such a way that no message can be transmitted unless all previous messages have been received properly. There are communications

processors, that handle the transmission and reception of messages and the retransmissions if necessary. This takes place at a lower level not seen and not affecting this Petri net. At this level, we assume that all messages arrive properly.

For further analysis on the Perti net and on other Petri nets modeling different input accessing methods refer to [5].

4. TIMING ANALYSIS

Suppose that the Master executes three times faster than the Slave. The Master process is checking for the presence of inputs three times faster than the Slave process. Figure 4 shows the requests for input of the Master and Slave processes to the operating system. These are denoted as markings on the Master and Slave axis respectively. Also shown are all the messages to synchronize a key that was pressed by the user at the Slave machine.

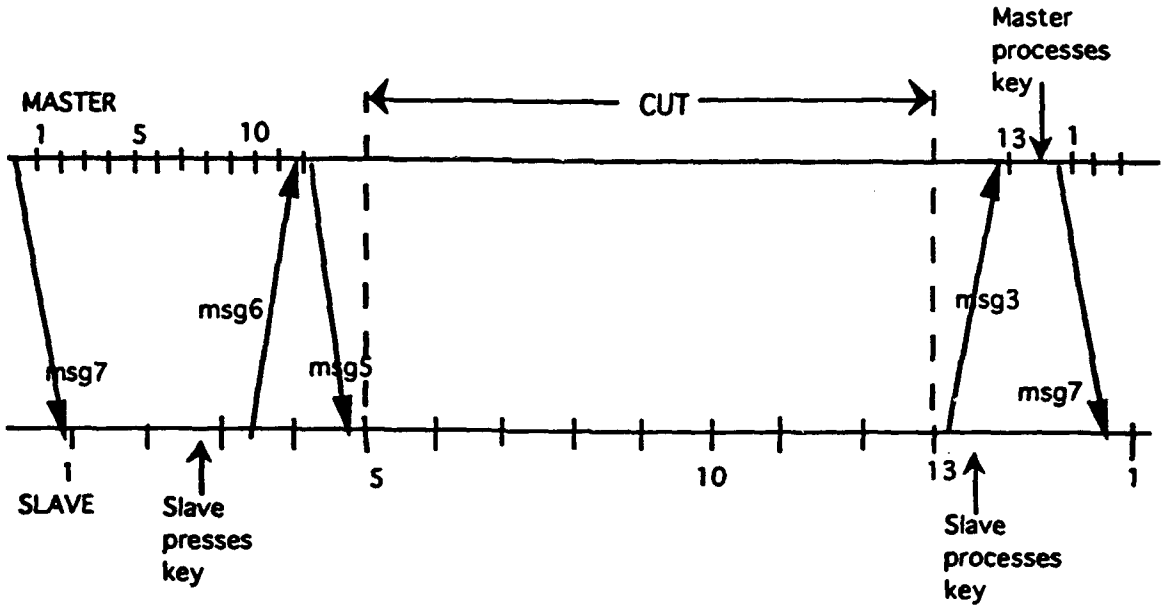


Figure 4 The Catch Up Time (CUT)

On the left side of figure 4, msg7 designates the end of the synchronization of the previous input. After the last synchronization, the Master and the Slave keep track of how many times the the application process asks the operating system to check for the presence of inputs, counters Cm and Cs respectively. Slave presses a key when Cs=2 and next time the Shell gets access (Cs becomes 3), msg6 is send to Master. Master receives msg6 at Cm=11 and gets processed when the Shell at Master

gets access (Cm becomes 12). Master compares counts and sends msg5 and Slave receives it at Cs=4 and will process it when Cs=5. Then the Slave has to catch up, and the Master has to wait for the Slave. At Cs=13 the Slave sends msg3 to the Master and stops its wait. On the next Master count, Cm=13, Master sends msg7 and a new synchronization phase starts.

It is noted that the Master process had to wait idle for an interval of 8*Ts, where *Ts* is the time interval between two consecutive times that the Slave checks for inputs, while the Slave catches up. This time interval will be the *Catch Up Time* (CUT) of the slower process.

The CUT must be such that the responsiveness of the system, from the moment that an input is entered till the moment the input is registered by the process, is in acceptable values for the users. That value will be called the *Maximum Allowable Wait Interval* (MAWI).

It is of interest to note that, even if the Master was executing idle loops waiting for an input, the Slave still has to do "idle catch up". In other words, the CUT of the Slave does not do any useful processing. The Slave's idle catch up cannot be eliminated since there is no way of knowing what the user process is executing without interfering with the process.

The more often the computers communicate, the closer they remain synchronized and the smaller the CUT required to regain synchronization on the next handshake. On the other hand, the more often the computers communicate, a higher bandwidth is required and less processing is achieved.

When some time passes without handshake (a timer expires), the Master computer initiates communication by introducing a null key which is removed after it gets synchronized. The users can set the value of this timer and tune he responsiveness of the system to their needs. In other words, they can set up the value for the MAWI. When the counter at the Master or the Slave approaches overflow a handshake with a null key is forced, initiated from the process whose counter approaches overflow.

The *worst case response* time of the system is equal to the CUT plus four Communication delays (ComDelay). Where ComDelay is the time required to

transmit a message from one site and is received on the other site. The Slave input gets synchronized with three messages to the Master and with four messages to the Slave. A Master input gets synchronized with two messages to the Master and with three messages to the Slave.

5. IMPORTANCE

The SPE technique helps to reduce communications in cases such as computer conferencing. Application programs can execute synchronously at geographically separated computers. Conferencing can take place through an application program. For example, if the conference is about the design of a building, the conference supporting program or application program could be AutoCad. The inputs to the program, originating from either computer according to an input accessing method, and some minimal synchronization information, such as the counts, are the only data that has to be communicated between the two computer systems. With existing methods of computer conferencing, execution takes place on one computer and every time the screen changes, the screens or the changes of the screens have to be communicated to the other computer. When the SPE technique is applied to computer conferencing, only the inputs to the program, such as the key strokes, have to be communicated.

Team work will be promoted even if its members are physically separated. Collaboration can take place with the use of existing, off the shelf software, and without any added hardware. Tutoring the use of software, debuging software, tutoring a subject through an application program, are some aspects that can take place remotely with the proposed approach. Application programs that produce many screens at a high frequency, are currently unable to be used remotely unless high bandwidth is provided. Using the proposed approach may allow communicating over existing telephone lines. Graphical programs ordinarily require considerable bandwidth. However, using the SPE approach allows them to run remotely over low speed lines.

Security is embedded in the communications when done with the proposed method. There are many circumstances where the inputs to generate the screens are not classified, while the generated screens (outputs) are classified. Many computer

users who would like to have a session through their computers, will be greatly benefitted from the results of this research.

REFERENCES

[1] J. L. Peterson, *Petri Net Theory and the Modeling of Systems*, Prentice- Hall, Inc., Englewood Cliffs, N.J., 1981

[2] J. E. Coolahan Jr. and N. Roussopoulos, "A timed Petri net methodology for specifing real-time system timing requirements," *International Workshop on Timed Petri Nets*, Torino, Italy, July 1985.

[3] S. M. Shatz and S. S. Yau, "The Application of Petri Nets to the Representation of Communication In Distributed Software Systems," *Proceedings of the 3rd International Conference on Distributed Computing Systems*, October 18-22, 1982.

[4] P. Merlin, "A Methodology for the Design and Implementation of Communication Protocols", *IEEE transactions on Communications*, Vol. 24, No. 6, June 1976, pp. 614-621.

[5]. Emmanouel Antonidakis, "Communications Reduction Using Simultaneous Program Execution," Ph.D. dissertation, Department of Electrical Computer and Systems Engineering, Boston University, Boston, MA, 1993.

MISSION

OF

ROME LABORATORY

Mission. The mission of Rome Laboratory is to advance the science and technologies of command, control, communications and intelligence and to transition them into systems to meet customer needs. To achieve this, Rome Lab:

a. Conducts vigorous research, development and test programs in all applicable technologies;

b. Transitions technology to current and future systems to improve operational capability, readiness, and supportability;

c. Provides a full range of technical support to Air Force Materiel Command product centers and other Air Force organizations;

d. Promotes transfer of technology to the private sector;

e. Maintains leading edge technological expertise in the areas of surveillance, communications, command and control, intelligence, reliability science, electro-magnetic technology, photonics, signal processing, and computational science.

The thrust areas of technical competence include: Surveillance, Communications, Command and Control, Intelligence, Signal Processing, Computer Science and Technology, Electromagnetic Technology, Photonics and Reliability Sciences.